Binding python to other languages (Fortran and C)

Overview

- One of the beauties of python is the ease with which you can bind it to low-level programming languages.
- Allows python to be a scripting interface on top of optimised CPU-intensive processing code.
- Examples are **CDAT** and **MetPy** developed by ECMWF.
- Numerous packages are available to do this.
- Here we present ***Pyfort***, ***F2PY*** for Fortran bindings and a quick look at C bindings (using ***SWIG***).

Locating and installing the packages

- You can freely **download** the packages at:
  - **Pyfort -** http://pyfortran.sourceforge.net
  - **F2PY -** http://cens.ioc.ee/projects/f2py2e

- **Installation:**
  - Both **Pyfort** and **F2PY** are now installed as part of CDAT and so is already available on a number of our linux machines under the directory:

```
<your_cdat>/bin/[pyfort|f2py]
```

*Much of the information in this document was stolen from:
http://www.prism.enes.org/WPs/WP4a/Slides/pyfort/pyfort.html

Pyfort Usage: Overview (1)

The interface to pyfort is relatively simple:

1. Pyfort takes a file or number of files holding Fortran functions and/or subroutines.

2. These are compiled and linked to a library.

3. The user then hand edits a Pyfort (**.pyf**) text file describing the interface to each function/subroutine.

Pyfort Usage: Overview (2)

4. The **pyfort** command is then run with the necessary arguments to produce some C code to describe the Fortran interface to python. Pyfort automatically compiles this C code into what is called a Python Extension Module (**.so**).

5. The Python Extension Module can then be imported directly into python with the functions/subroutines visible as module level python functions.

Pyfort Usage: Overview (3)

- Once you have created a **Python Extension Module** using Pyfort:
  - you will always have access to it at the Python level
  - from the user's perspective it is imported just like any other Python function.

Pyfort: A simple example (1)

- Below is a basic Fortran subroutine that has been connected to python. It demonstrates the use of the Pyfort interface without any complex code to confuse you:

- The **itimes.f** file contains the subroutine **itimes** which takes in two Numeric arrays (x and y) of length n and returns an array (w) of the same length where w(i)=x(i)*y(i).

```
        subroutine itimes(x,y,n,w)
        integer x(*)
        integer y(*)
        integer w(*)
        integer n
        integer i
        do 100 i=1,n
          w(i) = x(i) * y(i)
100     continue
        return
        end
```

Pyfort: A simple example (2)

- The “**itimes.f**” file is compiled as follows:

```
$ g77 -c itimes.f  # or use your compiler
```

- The compiled subroutine is then linked into a fortran library called **libitimes.a**:

```
$ ld -r -o libitimes.a itimes.o
```

Pyfort: A simple example (3)

- Must then write a Pyfort script declaring the parameters involved called **itimespyf.pyf**:

```
SUBROUTINE ITIMES(X, Y, N, W)
! times (x,y,n,x) sets (i)=x(i)*y(i), i=1,n
integer, intent(in):: x(n), y(n) ! must have
                                 size n
integer, intent(out)::w(n)
integer n
END SUBROUTINE itimes
```

- Finally, run Pyfort with the following arguments to produce the C code that glues it all together (this allows you to call the module and functions from python):

```
$ pyfort -c g77 -i -l./itimes itimespyf.pyf
```

Pyfort: A simple example (4)

- The output of this compilation was the production of a **Python Extension Module** called **itimespyf.so** located at:

  `./build/lib.linux-i686-2.2/itimespyf.so`

- You can then import this module directly into python and call the subroutine as python functions:

```
>>> import sys
>>> sys.path.append('build/lib.linux-i686-2.2')
>>> import itimespyf, Numeric
>>> x=Numeric.array([1,2,3])
>>> y=Numeric.array([4,5,6])
>>> n=len(x)
>>> print "itimes", x, y
itimes [1,2,3] [4,5,6]
>>> print testpyf.itimes(x,y,n)
[4,10,18]
```

F2PY Usage: Overview (1)

- F2PY demonstrates greater functionality than Pyfort, for example you can return character arrays, deal with **allocatable arrays** and **common blocks**, which pyfort does not allow.

- The F2PY interface is potentially simpler than that for Pyfort, but there are various methods you can choose from. The F2PY documentation takes you through these methods.

F2PY: A simple example (1)

1. Create a fortran file such as **hello.f**:

```
C File hello.f
      subroutine foo (a)
      integer a
      print*, "Hello from Fortran!"
      print*, "a=",a
      end
```

2. Run F2PY on the file:

```
$ f2py -c -m hello hello
```

F2PY: A simple example (2)

- Run python and import the module, then call the subroutine as a function:

```
$ python
>>>   import hello
>>>   hello.foo(34)
‘Hello from Fortran!’
a= 34
```

Choosing between Pyfort and F2PY

- **F2PY** is the more comprehensive of the two packages (**providing support for returning character arrays, simple F90 modules, common blocks, callbacks and allocatable arrays**) but if pyfort does what you want, it may be easier to get to grips with.

- Both **Pyfort and F2PY are useful tools** and deciding on which one to use will depend on a number of issues. In theory, using either tool should be a quick (less than 1 hour) job but determining the duration will depend on issues such as:

How to choose

- Which package am I experienced with?
- Which package is available already on my platform?
- How long does it take to install (if not already present)?
- Which Fortran compiler am I using?
- Can I get away with the quick F2PY solution that involves no hand editing of files?
- Do I need to return character arrays from my subroutine (in which case you need to use F2PY)?
- Am I using callbacks (need F2PY again)?
- Do I need to handle F90 modules (need F2PY again)?
- Do I need to use Common Blocks (need F2PY again)?
- Does my code use Allocatable Arrays (need F2PY again)?

Python to C/C++ binding with SWIG
(Simplified Wrapper and Interface Generator)

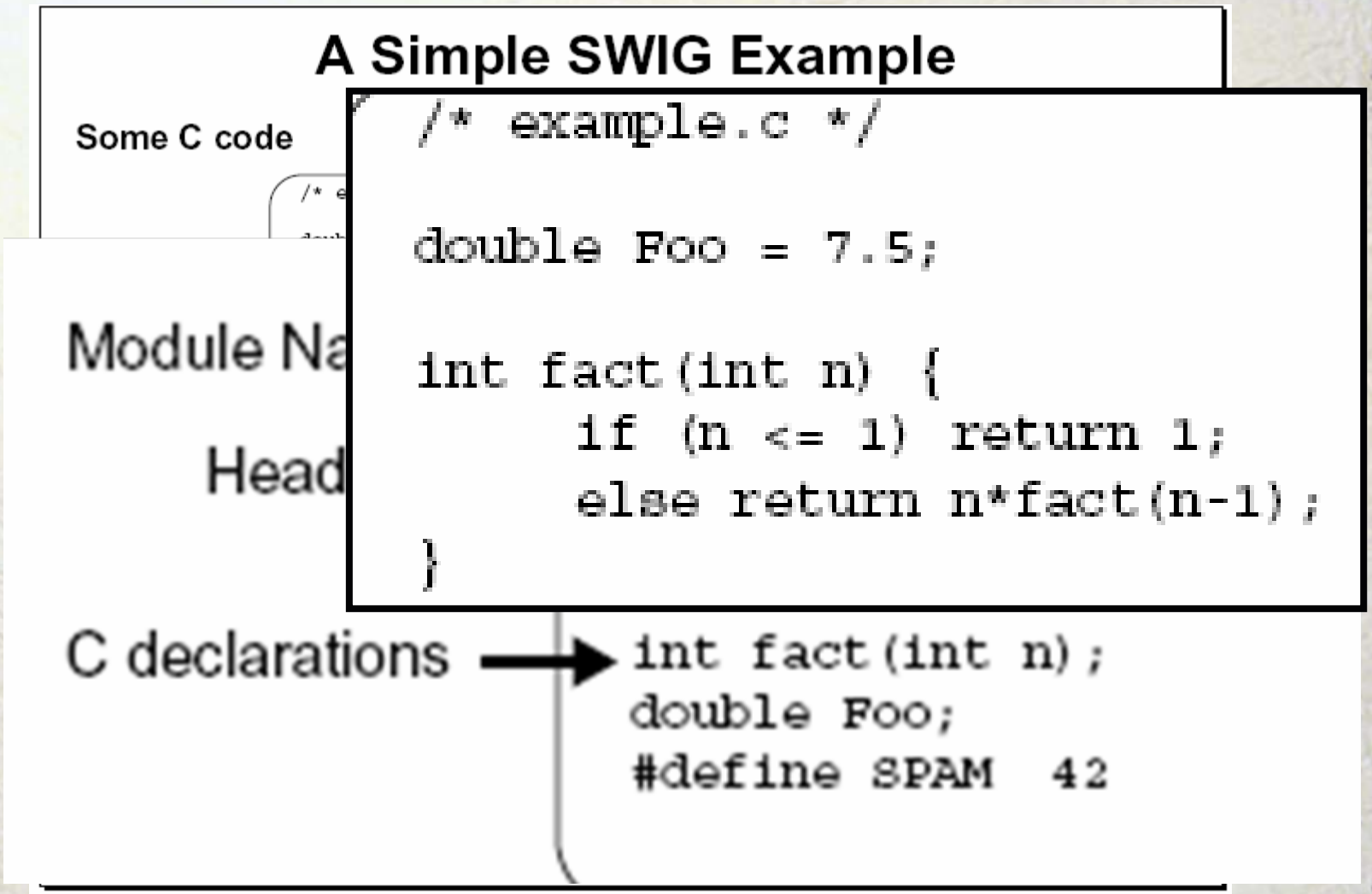
- SWIG is a useful tool that allows you to create python wrappers for C code with very little knowledge of the Python C API (but it might not always work).

- It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that scripting languages need to access the underlying C/C++ code.

- The SWIG interface compiler also connects programmes written in C and C++ with other languages including Perl, Ruby, and Tcl.

*Much of the information in this document was stolen from the official python documentation at:
http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf

SWIG Example (1)



*Much of the information in this document was stolen from the official python documentation at:
http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf

SWIG Example (2)

A Simple SWIG Example (cont...)

Building a Python Interface

```
% swig -python example.i
Generating wrappers for Python
% cc -c example.c example_wrap.c \
        -I/usr/local/include/python1.5 \
        -I/usr/local/lib/python1.5/config
% ld -shared example.o example_wrap.o -o examplemodule.so
```

```
24
>>> print example.cvar.Foo
7.5
>>> print example.SPAM
42
```


```
>>> print example.SPAM
42
```


*Much of the information in this document was stolen from the official python documentation at:

http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf